A Technical Analysis and Implementation Strategy for the Adobe "Connecting the Dots" Hackathon

Section 1: Feasibility Analysis: Your Laptop vs. The Challenge Environment

This section provides a comprehensive analysis of the feasibility of developing the hackathon solution on the specified local hardware. It compares the user's laptop specifications against the official evaluation environment, identifies potential bottlenecks, and delivers a conclusive verdict with a strategic path forward.

1.1 Deconstructing the Hardware Requirements

A successful engineering endeavor begins with a clear understanding of the operational environment. The hackathon presents two distinct environments: the local development machine and the final evaluation platform. A direct comparison reveals the critical constraints and opportunities.

- **Local Development Machine:**
  - **CPU:** AMD Ryzen 5 5000 series (6-core, 12-thread)
  - **RAM:** 8 GB
  - **GPU:** NVIDIA GTX series
- Hackathon Evaluation Environment [1]:
  - **CPU:** 8 CPUs (amd64 / x86_64 architecture)
  - **RAM:** 16 GB
  - **GPU:** Not available (No GPU dependencies rule)

The comparison yields immediate conclusions. The local CPU, a modern 6-core processor, is more than sufficient for the tasks at hand and is comparable in performance class to the 8-CPU evaluation environment. The most significant

divergence is the available system memory, with the evaluation environment offering double the RAM of the local machine. The strict no GPU dependencies rule is a great equalizer; it renders the local NVIDIA GPU irrelevant for the final submission but presents a potential strategic asset for specific, offline development tasks.

1.2 The 8GB RAM Bottleneck: A Critical Assessment

The 8 GB of RAM on the local development machine constitutes the primary technical challenge. While the final solution will run in a more generous 16 GB environment, the entire development, testing, and debugging cycle must be managed within this tighter memory budget. This constraint will exert pressure at several key stages of the workflow:

- **PDF Parsing and Data Loading:** While modern PDF libraries are optimized for performance, loading and parsing multiple large documents, especially those rich in images or complex vector graphics, can lead to significant memory allocation. A library like PyMuPDF is highly memory-efficient, but processing the maximum collection of 10 PDFs for Round 1B simultaneously could strain an 8 GB system.[1]
- **ML Model Loading:** This is the most significant potential consumer of RAM. For Round 1B, an embedding model must be loaded into memory. A popular and suitable model like sentence-transformers/all-MiniLM-L6-v2 has a disk size of approximately 80 MB but can expand to several hundred megabytes when loaded into a framework like PyTorch.[3] This single operation will consume a notable fraction of the available 8 GB.
- **In-Memory Data Structures:** The core of the Round 1B solution involves generating and comparing embeddings. Storing the embeddings for all text chunks from up to 10 PDFs (each up to 50 pages) in memory requires careful management. If not handled efficiently, this collection of floating-point vectors can grow to hundreds of megabytes.

The 8 GB RAM limitation is not an insurmountable obstacle but a design constraint that forces discipline and optimization from the outset. This is, in fact, an advantage. By architecting the solution to perform well under this pressure, it will be exceptionally efficient in the 16 GB evaluation environment. Key mitigation strategies include:

- **Memory-Efficient Coding:** Employing Python best practices such as using generators to process data lazily, handling files serially rather than in parallel, and explicitly freeing memory of large, temporary variables using del var followed by a

call to the garbage collector (gc.collect()).
- **Optimized Data Types:** Using libraries like NumPy and specifying memory-efficient data types (e.g., numpy.float32 instead of the default float64 for embeddings) can halve the memory footprint of large arrays.

1.3 CPU and GPU Considerations

The processing power of the local machine is well-aligned with the challenge requirements.

- **CPU:** The 6-core, 12-thread AMD Ryzen 5 CPU is a powerful and modern processor. The tasks outlined—rule-based parsing in Round 1A and optimized vector mathematics in Round 1B—are not typically CPU-bound on such hardware. The 10-second and 60-second execution time limits are aggressive but achievable through algorithmic efficiency rather than raw processing power.[1]
- **GPU:** The No GPU dependencies rule is absolute for the final Docker submission.[1] However, the local NVIDIA GTX GPU can be a strategic tool during the development phase for optional, advanced explorations. For instance, if one were to attempt to distill a custom, highly-specialized embedding model from a larger one using a library like
  Model2Vec, the GPU could accelerate this one-time, offline training process dramatically.[5] This is an advanced path but illustrates how the local GPU can still provide value without violating the submission rules.

1.4 Verdict: Feasibility and Path to Success

The verdict is clear: Building the required system on the specified laptop is challenging but unequivocally feasible.

Success is contingent upon adopting a disciplined, memory-conscious development methodology. The 8 GB RAM limitation is the most significant hurdle, but it should be viewed as a forcing function for good engineering practices. It necessitates a focus on efficient algorithms, careful library selection, and robust memory management. Ultimately, the development process itself, rather than the final execution, is the primary challenge. By building a modular system and testing components in isolation,

the memory constraints can be effectively managed. A solution engineered to be lean and efficient on 8 GB of RAM will not only work but will excel with the additional headroom provided by the 16 GB evaluation environment.

Section 2: A Strategic Blueprint for Round 1A: PDF Structure Extraction

This section provides a detailed technical architecture for building a high-performance, high-accuracy PDF outline extractor for Round 1A. The proposed solution is designed to meet all hackathon constraints while delivering competitive results.

2.1 The Core Challenge: Beyond Simple Text Extraction

The mission for Round 1A is to extract a document's hierarchical structure (Title, H1, H2, H3) into a JSON format.[1] The crucial piece of guidance provided is, "Don't rely solely on font sizes for heading level determination".[1] This single sentence elevates the problem from a trivial parsing task to a nuanced document intelligence challenge. It signals that a robust solution cannot rely on a single, simple heuristic.

Instead, the problem must be approached as one of structured data extraction. The goal is to identify text fragments that serve a structural purpose. This requires extracting not just the text itself, but its associated metadata: font properties (size, name, weight), positional information (x-y coordinates, indentation), and textual patterns.[2] A successful solution will be a heuristic engine that weighs multiple features to classify each piece of text.

2.2 Selecting the Right Tool: A Comparative Analysis of Python PDF Libraries

The choice of a PDF processing library is the most critical foundational decision for Round 1A. The ideal library must be fast, lightweight, and provide low-level access to

the rich metadata embedded within a PDF. A comparative analysis points to one clear winner: **PyMuPDF (Fitz)**.

PyMuPDF consistently benchmarks as the fastest Python PDF library, a critical factor for meeting the ≤10 second execution time limit.[8] More importantly, it provides the

page.get_text("dict") method, which is perfectly suited for this task. This method returns a deeply nested dictionary containing all text blocks, lines, and spans on a page, complete with their precise bounding boxes (bbox), font names, font sizes, and a flags attribute that encodes properties like bold and italic.[10] This structured output is the ideal raw material for the heuristic engine.

Table 2.1: Python PDF Library Head-to-Head Comparison for Round 1A

Library	Speed / Performance	Metadata Richness (Font, Position, Flags)	Memory Footprint	Ease of Use (for this task)
PyMuPDF (Fitz)	**Excellent** (Fastest C-based backend) [8]	**Excellent** (Provides size, name, flags, bbox for every span via get_text("dict")) [11]	**Low**	**High** (The "dict" output is directly consumable for heuristic analysis)
pdfplumber	Good (Built on pdfminer.six) [13]	Good (Provides access to char-level data including fontname and size) [13]	Medium	Medium (Requires more iteration to assemble the necessary features)
pdfminer.six	Moderate [9]	Good (Can extract detailed font and layout info, but API is more complex) [6]	Medium	Low (API is less intuitive for this specific structured extraction)
PyPDF2 / pypdf	Slowest (Pure Python) [8]	Poor (Primarily extracts raw text; limited and	Low	Very Low (Not suitable for metadata-driven

		difficult access to font/position metadata) [15]		analysis)

The analysis in Table 2.1 confirms that PyMuPDF offers the best combination of speed and detailed metadata access, making it the superior choice for this challenge.

2.3 Implementing a Robust Heading Detection Engine

The core of the solution is a multi-featured heuristic engine. The approach is to perform a statistical analysis of the document first to establish a baseline, then classify text spans based on how they deviate from that baseline. This avoids brittle, hard-coded rules and adapts to the specific styling of each PDF.

The process involves iterating through the text blocks returned by page.get_text("dict") and assigning a "heading score" based on a weighted combination of the following features:

1. **Font Size:** A primary indicator. The engine should first perform a pass to find the most common font size in the document, which typically represents the body text. Headings are then identified as statistical outliers with significantly larger sizes.[6]
2. **Font Weight (Bold):** A very strong signal. The flags key in a PyMuPDF span dictionary can be bitwise-checked against the TEXT_FONT_BOLD constant (16) to detect bold text.[12]
3. **Font Name:** Headings may use a distinct font family (e.g., a sans-serif heading font like 'Arial-Bold' with a serif body font like 'Times New Roman'). The engine can track font names and flag changes.[17]
4. **Positional Data:**
    - **Indentation:** The x0 coordinate of a line's bounding box indicates its indentation. Headings are typically left-aligned with minimal indentation.
    - **Vertical Spacing:** A line with significantly more white space above it than below it is likely a heading. This can be calculated by comparing the y coordinates of consecutive text blocks.[7]
5. **Textual Patterns:** Regular expressions can be used to detect common heading formats, such as numbered lists (1. Introduction, 2.1. Methods), alphabetic lists (A. Appendix), or lines written in ALL CAPS.[18]
6. **Line Length and Punctuation:** Headings are typically short, concise phrases and rarely end with a period. Lines that are long or end with sentence-terminating

punctuation are unlikely to be headings.

These features are then combined to classify each heading level hierarchically.

Table 2.2: Heuristics for Hierarchical Heading Classification

Heading Level	Primary Indicators	Secondary Indicators	Tertiary Indicators	Negative Indicators
Title	Largest font size in the document; often on the first page.	Centered horizontally; significant vertical space above and below.	Often uses a unique font name.	Appears on subsequent pages.
H1	Second-largest font size category; significantly larger than body text.	Typically bold (TEXT_FONT_BOLD flag is set).	Low indentation (x0 is small); often has a numeric prefix (e.g., "1.").	Line is very long; ends with a period.
H2	Third-largest font size category; noticeably larger than body text.	Typically bold; may be italic (TEXT_FONT_ITALIC flag).	Indented slightly more than H1; often has a two-level numeric prefix (e.g., "1.1").	Follows an H3 without an intervening H1/H2.
H3	A font size that is slightly larger than body text, or is body size but bold.	Often bold or italic.	Indented more than H2; often has a three-level numeric prefix (e.g., "1.1.1").	Text is part of a multi-line paragraph.

2.4 Workflow and Implementation Guide

The implementation follows a logical, multi-pass process:

1. **Initialization:** Open the target PDF using pymupdf.open(filepath).

2. **Global Statistics Pass:** Iterate through all pages and all text spans from page.get_text("dict") to compute document-wide statistics. This includes creating a frequency distribution of all font sizes and names to identify the most common (body text) properties.
3. **Boilerplate Removal:** Identify and flag running headers and footers. A robust method is to identify text strings that appear in nearly the same bounding box (bbox) across a high percentage (>50%) of pages. These flagged text blocks are then ignored during classification.[19]
4. **Classification Pass:** Re-iterate through the pages. For each non-boilerplate text line, apply the heuristic engine defined in Table 2.2, using the global statistics as a baseline. Assign a potential heading level (H1, H2, H3, Title) or 'body' to each line.
5. **Hierarchical Structuring:** Post-process the list of classified headings to enforce logical order. For example, if an H3 is detected after an H1, any intermediate text that was weakly classified as 'body' could be re-evaluated. The final outline must maintain a valid hierarchy (an H3 must be a child of an H2, which must be a child of an H1).
6. **JSON Output Generation:** Format the final, cleaned, and structured list of headings into the precise JSON format specified in the challenge document, including level, text, and page number.[1]

2.5 Meeting the Constraints

The proposed architecture is explicitly designed to meet all Round 1A constraints:

- **Execution Time (≤10 seconds):** PyMuPDF is exceptionally fast. The two-pass heuristic engine involves simple dictionary lookups and arithmetic, which are computationally trivial. A 50-page PDF can be processed well within the 10-second limit.[8]
- **Model Size (≤200 MB):** This solution is purely algorithmic and uses no machine learning models. The model size is therefore 0 MB, easily satisfying the constraint.[1]
- **Offline and CPU-only:** The solution relies only on the PyMuPDF library and standard Python libraries. It requires no network access and runs entirely on the CPU, making it fully compliant.

Section 3: Architecting the Solution for Round 1B: Persona-Driven Intelligence

Round 1B elevates the challenge from structural analysis to semantic understanding. The task is to build an "intelligent document analyst" that extracts and ranks the most relevant sections from a collection of PDFs based on a user persona and their specific goal.[1] This is a classic semantic search problem, and success requires a different, yet equally efficient, set of tools and strategies.

3.1 The Problem Space: From Structure to Semantics

The core of Round 1B is to measure the semantic similarity between a query (the persona and job-to-be-done) and a corpus of documents (the PDF collection). This process can be broken down into three fundamental stages of a modern search pipeline [22]:

1. **Content Ingestion and Chunking:** Processing the raw PDFs and breaking them down into smaller, semantically meaningful units of text ("chunks").
2. **Semantic Representation (Embedding):** Using a deep learning model to convert the text of the query and each chunk into a high-dimensional numerical vector, or "embedding." These embeddings capture the semantic meaning of the text.
3. **Relevance Ranking:** Calculating the mathematical similarity (typically cosine similarity) between the query's embedding and all chunk embeddings, and ranking the chunks from most to least similar.

The stringent constraints—CPU-only, offline, ≤1GB model size, and ≤60 seconds processing time—are the primary forces shaping every architectural decision in this pipeline.[1]

3.2 Foundational Step: Intelligent Document Chunking

The quality of a semantic search system is profoundly influenced by its chunking

strategy. Poorly formed chunks that split coherent ideas or mix unrelated topics will lead to inaccurate and irrelevant results.[24] While common methods like fixed-size or recursive character splitting exist, a far superior approach is available by leveraging the work from Round 1A.

The optimal strategy is **content-aware chunking**, using the document's own structure as the guide.[26] The structured outline of Titles, H1s, H2s, and H3s extracted in Round 1A provides natural semantic boundaries. The implementation is as follows:

- A "chunk" is defined as a heading (e.g., "2.1 Methodology") plus all the text that follows it, up to the next heading of the same or a higher level.
- This ensures that each chunk is a self-contained, logical unit of the document. For example, the entire discussion of a specific method or result is kept together.
- This approach directly embodies the hackathon's theme of "Connecting the Dots," as the structural dots identified in 1A are used to create the semantic units for 1B. The quality of retrieval in this round is therefore directly dependent on the accuracy of the heading extraction in the previous round.

3.3 The Engine of Intelligence: Choosing a Lightweight Embedding Model

The selection of the embedding model is the most critical decision for Round 1B. It must be small enough to fit within the ≤1GB size limit and fast enough to run on a CPU, yet powerful enough to generate high-quality semantic embeddings. The ideal tools for this are **Sentence Transformer** models, which are specifically fine-tuned for producing sentence and paragraph-level embeddings for similarity tasks.[27]

The following table compares several excellent, lightweight candidates suitable for this challenge.

Table 3.1: Comparison of Lightweight Sentence Transformer Models for Round 1B

Model Name	Disk Size (Approx.)	Embedding Dimension	CPU Speed	MTEB Score (Performance)	Key Characteristics

sentence-transformers/all-MiniLM-L6-v2	**80 MB**	**384**	**Excellent**	**Good (56.09)** [29]	The de-facto standard for a balanced, high-quality, and fast model. Excellent for general-purpose semantic search. [3]
BAAI/bge-small-en-v1.5	130 MB	384	Very Good	Very Good	A strong performer, slightly larger than MiniLM but often with a small performance edge in retrieval tasks. [31]
MinishLab/potion-base-32M (Model2Vec)	32 MB	256	**Excellent**	Good (MTEB depends on distillation source)	An extremely small and fast static model distilled from a larger transformer. A more advanced but potentially very efficient option. [5]

Recommendation: For this hackathon, **sentence-transformers/all-MiniLM-L6-v2** is the recommended model. It offers the best all-around balance of small size, high speed, and proven, robust performance. Its 80 MB footprint is trivial for the 1 GB limit and will be manageable during development on an 8 GB machine. Its widespread use and strong documentation make it a reliable and low-risk choice.[30]

3.4 The Ranking Mechanism: Implementing Efficient Semantic Search

The end-to-end workflow for Round 1B combines the components discussed above into an efficient pipeline:

1. **Input Processing:** The persona and job-to-be-done text inputs are not treated as separate entities. They should be concatenated into a single, rich query string. For example: f"As a {persona}, my goal is to {job_to_be_done}". This creates a unified semantic query vector that captures the full context of the user's intent without requiring complex, brittle logic.[32]
2. **Embedding Generation:**
    - The chosen Sentence Transformer model (all-MiniLM-L6-v2) is loaded into memory once at the start of the process.
    - The unified query string is passed to model.encode() to generate a single query embedding.
    - The solution then iterates through all documents, parsing them with the Round 1A logic to generate content-aware chunks. Each chunk's text is encoded to produce a chunk embedding.
    - All chunk embeddings are stored in a single numpy array for efficient processing.
3. **Similarity Calculation and Ranking:**
    - The cosine similarity between the query embedding and the entire array of chunk embeddings is calculated. This is a single, highly optimized matrix operation. The sentence_transformers.util.cos_sim function is ideal for this, as it leverages PyTorch for fast computation even on a CPU.[23]
    - The results are a list of similarity scores, one for each chunk.
4. **Output Formatting:**
    - The chunks are sorted in descending order based on their similarity scores.
    - An importance_rank is assigned to each chunk based on its position in the sorted list.
    - The top-ranked chunks are formatted into the final JSON structure required by the challenge, including metadata like the source document and page number. The "Refined Text" for the "Sub-section Analysis" can be generated by extracting the first few sentences from the corresponding top-ranked chunk.

3.5 Adhering to Round 1B Constraints

This architecture is designed from the ground up to respect the challenge's constraints:

- **Processing Time (≤60 seconds):** The pipeline is highly optimized. PyMuPDF is fast for parsing. all-MiniLM-L6-v2 is fast for encoding on a CPU. Cosine similarity calculation on a few thousand chunks using numpy or torch is near-instantaneous. The entire process for 3-10 documents will be well within the 60-second limit.
- **Model Size (≤1 GB):** The recommended all-MiniLM-L6-v2 model is only ~80 MB, leaving ample room for all other dependencies within the 1 GB limit.
- **CPU-only & Offline:** The entire technology stack—PyMuPDF, Sentence Transformers (with PyTorch configured for CPU), and NumPy—is fully CPU-compatible and works entirely offline once the model is downloaded and packaged.

Section 4: Deployment and Final Submission: The Docker Workflow

A brilliant algorithm is worthless in a hackathon if it cannot be run by the judges. The Docker container is the sole deliverable and the only way the solution will be evaluated. Therefore, creating a correct, robust, and compliant Dockerfile is as critical as writing the Python code itself.

4.1 Crafting the Definitive Dockerfile

The Dockerfile must create a self-contained, executable environment that adheres to all platform and network constraints. The following provides a complete, commented template for the solution.

Dockerfile

```
# Stage 1: Base Image and Platform Specification
# Use an official Python slim image for a smaller footprint.
# CRITICAL: Explicitly set the platform to match the evaluation environment.
FROM --platform=linux/amd64 python:3.9-slim

# Set the working directory inside the container
WORKDIR /app

# Create a directory for the ML model
RUN mkdir -p /app/models

# Pre-download the model locally and copy it into the image.
# This step ensures the container is fully self-contained and works offline.
# Assumes the 'all-MiniLM-L6-v2' model was downloaded to a 'models' directory locally.
COPY./models/all-MiniLM-L6-v2 /app/models/all-MiniLM-L6-v2

# Copy the requirements file before copying the rest of the code
# This leverages Docker's layer caching. The dependencies won't be re-installed
# unless the requirements.txt file changes.
COPY requirements.txt.

# Install dependencies. The --no-cache-dir flag reduces image size.
# Ensure 'torch' is installed with CPU-only support if possible to save space.
RUN pip install --no-cache-dir -r requirements.txt

# Copy the application source code into the container
COPY./src./src

# Define the command to run the application.
# This script should be designed to read from /app/input and write to /app/output
# as specified in the hackathon rules.
CMD ["python", "src/main.py"]
```

This Dockerfile is structured for efficiency and compliance. It correctly sets the platform, copies the pre-downloaded model to ensure offline capability, and uses layer caching for faster builds during development.

4.2 Ensuring Offline and Platform Compatibility

Two rules are paramount for the Docker submission:

1. **Platform:** The FROM --platform=linux/amd64 instruction in the Dockerfile and the --platform linux/amd64 flag in the docker build command are mandatory. They ensure that the image is built for the correct CPU architecture used by the judges.[1]
2. **Offline Execution:** The docker run command will use the --network none flag, completely isolating the container from the internet.[1] This has a critical implication: any dependency, especially ML models typically downloaded on-the-fly by libraries like sentence-transformers, must be pre-downloaded and included in the Docker image during the build phase. The workflow must be:
   a. On the local machine, run a simple Python script to download and cache the model: SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2').
   b. Locate the cached model files (e.g., in ~/.cache/huggingface/hub/).
   c. Copy these files into the project directory (e.g., ./models/all-MiniLM-L6-v2).
   d. The Dockerfile then COPYs this local directory into the image.
   e. The Python code must then be pointed to load the model from this specific path within the container (e.g., /app/models/all-MiniLM-L6-v2).

4.3 A Complete Build-and-Run Protocol

To eliminate any surprises during evaluation, the solution must be tested locally using the exact commands the judges will use. This provides a complete simulation of the final run.

1. **Create Input/Output Directories:**
   Bash
   ```
   mkdir -p input output
   cp /path/to/sample.pdf./input/
   ```

2. **Build the Docker Image:**
   Bash
   ```
   docker build --platform linux/amd64 -t my_solution:latest.
   ```

3. **Run the Container:**
   Bash
   ```
   docker run --rm -v "$(pwd)/input:/app/input" -v "$(pwd)/output:/app/output" --network
   ```

```
none my_solution:latest
```

After the command completes, the output directory should contain the generated JSON files. This protocol provides a foolproof method for local validation.

Section 5: Concluding Strategy and Pro-Tips for Competitive Edge

Beyond the core technical implementation, strategic decisions regarding architecture, bonus objectives, and project organization can significantly impact the final score and distinguish a submission from the competition.

5.1 A Unified and Modular Architecture

As explicitly recommended in the challenge document, a modular code structure is essential.[1] This not only aids in development and debugging, especially on a memory-constrained machine, but also demonstrates good software engineering practice. The two rounds are not isolated; they form a pipeline where the output of 1A can be the input for 1B. A unified architecture reflects this understanding.

A recommended project structure:

```
/project-root/
├── Dockerfile
├── README.md
├── requirements.txt
├── approach_explanation.md  # For Round 1B
├── models/
│   └── all-MiniLM-L6-v2/    # Pre-downloaded model files
```

```
│       ├──...
└── src/
    ├── main.py             # Main script, entry point for Docker
    ├── config.py           # Configuration, file paths, model names
    ├── pdf_parser.py       # Core logic for Round 1A (heading extraction)
    ├── semantic_searcher.py # Core logic for Round 1B (embedding and ranking)
    └── utils.py            # Shared utility functions (e.g., JSON handling)
```

In this structure, main.py orchestrates the process, calling the pdf_parser to get the structured outline and then feeding that structure and the text into the semantic_searcher for Round 1B's task.

5.2 Strategies for Earning Bonus Points

- **Multilingual Handling (Round 1A Bonus):** The proposed heuristic engine for Round 1A is surprisingly robust for multilingual documents. Features like font size, font weight, and vertical spacing are language-agnostic. Textual patterns like numeric prefixes (1., 1.1.) are also common across many languages. The primary challenge would be languages with fundamentally different structures, like CJK (Chinese, Japanese, Korean). However, PyMuPDF has excellent support for extracting CJK text [35], so the raw data would be available. The engine could be enhanced with language detection to switch to different regex patterns if needed, making the bonus achievable.
- **Demonstrating Deeper Research:** To stand out, the README.md and approach_explanation.md can briefly mention and justify the rejection of more complex methods in favor of efficiency. For example, one could mention exploring advanced layout analysis techniques like using clustering algorithms (e.g., DBSCAN) on text block coordinates to more robustly identify columns or marginalia, but concluding that the multi-pass heuristic approach provided a better balance of accuracy and speed for this specific challenge.[19] This demonstrates a depth of research that goes beyond the first-pass solution.

5.3 Final Checklist and Parting Advice

Before submission, ensure all deliverables are present and polished:

- **Git Project:** A private Git repository containing all code and documentation.
- **Dockerfile:** A working, commented Dockerfile in the root directory.
- **README.md:** A comprehensive README explaining the approach, libraries used, and the build/run protocol.
- **approach_explanation.md:** A dedicated 300-500 word document for Round 1B explaining the methodology.[1]
- **Dependencies:** A clean requirements.txt file.
- **Offline Assets:** The ML model files are included within the project structure.

In a hackathon with tight constraints, victory often goes not to the most complex solution, but to the most robust and efficient one. The architecture proposed in this report prioritizes fundamentals: fast and reliable parsing, a well-chosen lightweight model, and an efficient algorithmic pipeline. By focusing on building a clean, well-engineered, and compliant system, a team can confidently meet the challenge's requirements and produce a high-quality, competitive submission.

Works cited

1. 6874faecd848a_Adobe_India_Hackathon_-_Challenge_Doc.pdf
2. A Guide to PDF Extraction Libraries in Python - Metric Coders, accessed on July 22, 2025, https://www.metriccoders.com/post/a-guide-to-pdf-extraction-libraries-in-python
3. sentence-transformers/all-MiniLM-L6-v2 - Hugging Face, accessed on July 22, 2025, https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
4. Unlocking the Power of Sentence Embeddings with all-MiniLM-L6-v2 - Medium, accessed on July 22, 2025, https://medium.com/@rahultiwari065/unlocking-the-power-of-sentence-embeddings-with-all-minilm-l6-v2-7d6589a5f0aa
5. MinishLab/model2vec: Fast State-of-the-Art Static Embeddings - GitHub, accessed on July 22, 2025, https://github.com/MinishLab/model2vec
6. How to extract text from PDF on the basis of font size using python libraries. - Medium, accessed on July 22, 2025, https://medium.com/@i190712/how-to-extract-text-from-pdf-on-the-basis-of-font-size-using-python-libraries-e931e583749b
7. PDF header and paragraph detection · jsvine pdfplumber · Discussion #868 - GitHub, accessed on July 22, 2025, https://github.com/jsvine/pdfplumber/discussions/868
8. Features Comparison - PyMuPDF 1.26.3 documentation, accessed on July 22, 2025, https://pymupdf.readthedocs.io/en/latest/about.html
9. Comparing 4 methods for pdf text extraction in python | by Jeanna Schoonmaker | Social Impact Analytics | Medium, accessed on July 22, 2025,

https://medium.com/social-impact-analytics/comparing-4-methods-for-pdf-text-extraction-in-python-fd34531034f
10. Tutorial - PyMuPDF 1.26.3 documentation, accessed on July 22, 2025, https://pymupdf.readthedocs.io/en/latest/tutorial.html
11. Appendix 1: Details on Text Extraction - PyMuPDF 1.26.3 documentation, accessed on July 22, 2025, https://pymupdf.readthedocs.io/en/latest/app1.html
12. Constants and Enumerations - PyMuPDF 1.26.3 documentation, accessed on July 22, 2025, https://pymupdf.readthedocs.io/en/latest/vars.html
13. jsvine/pdfplumber: Plumb a PDF for detailed information about each char, rectangle, line, et cetera — and easily extract text and tables. - GitHub, accessed on July 22, 2025, https://github.com/jsvine/pdfplumber
14. PDFminer: extract text with its font information [duplicate] - Stack Overflow, accessed on July 22, 2025, https://stackoverflow.com/questions/34606382/pdfminer-extract-text-with-its-font-information
15. Extract Text from a PDF — pypdf 3.15.2 documentation - Read the Docs, accessed on July 22, 2025, https://pypdf.readthedocs.io/en/3.15.2/user/extract-text.html
16. How to Extract Text from a PDF Using PyMuPDF and Python | by Neurond AI | Medium, accessed on July 22, 2025, https://neurondai.medium.com/how-to-extract-text-from-a-pdf-using-pymupdf-and-python-caa8487cf9d
17. How to Extract Text from PDFs Without Losing Formatting (Italics, Bold, Underlines) - Reddit, accessed on July 22, 2025, https://www.reddit.com/r/pdf/comments/1i1j8r3/how_to_extract_text_from_pdfs_without_losing/
18. How to extract text under specific headings from a pdf? - Stack Overflow, accessed on July 22, 2025, https://stackoverflow.com/questions/48107611/how-to-extract-text-under-specific-headings-from-a-pdf
19. Is there a way to delete headers/footers in PDF documents? #2259 - GitHub, accessed on July 22, 2025, https://github.com/pymupdf/PyMuPDF/discussions/2259
20. Excluding the Header and Footer Contents of a page of a PDF file while extracting text?, accessed on July 22, 2025, https://stackoverflow.com/questions/52039847/excluding-the-header-and-footer-contents-of-a-page-of-a-pdf-file-while-extractin
21. Extracting headers and footers (text repeated on every page) from documents, accessed on July 22, 2025, https://stackoverflow.com/questions/64958301/extracting-headers-and-footers-text-repeated-on-every-page-from-documents
22. How to Implement Semantic Search in Python Step by Step - TiDB, accessed on July 22, 2025, https://www.pingcap.com/article/semantic-search-python-step-by-step/
23. A simple semantic search with Python and FastAPI | by Udit Rawat | AgentsOps - Medium, accessed on July 22, 2025,

https://medium.com/agentsops/a-simple-semantic-search-with-python-and-fastapi-f80f2b785086
24. Chunking Strategies for LLM Applications - Pinecone, accessed on July 22, 2025, https://www.pinecone.io/learn/chunking-strategies/
25. What chunking strategies work best for document indexing? - Milvus, accessed on July 22, 2025, https://milvus.io/ai-quick-reference/what-chunking-strategies-work-best-for-document-indexing
26. Chunking strategies for RAG tutorial using Granite - IBM, accessed on July 22, 2025, https://www.ibm.com/think/tutorials/chunking-strategies-for-rag-with-langchain-watsonx-ai
27. sentence-transformers - PyPI, accessed on July 22, 2025, https://pypi.org/project/sentence-transformers/
28. What are the best open-source libraries for semantic search? - Milvus, accessed on July 22, 2025, https://milvus.io/ai-quick-reference/what-are-the-best-opensource-libraries-for-semantic-search
29. Model2Vec: Distill a Small Fast Model from any Sentence Transformer - Hugging Face, accessed on July 22, 2025, https://huggingface.co/blog/Pringled/model2vec
30. What are some popular pre-trained Sentence Transformer models and how do they differ (for example, all-MiniLM-L6-v2 vs all-mpnet-base-v2)? - Milvus, accessed on July 22, 2025, https://milvus.io/ai-quick-reference/what-are-some-popular-pretrained-sentence-transformer-models-and-how-do-they-differ-for-example-allminilml6v2-vs-allmpnetbasev2
31. Which model should i use to embedding 150-200 words sentence? : r/Rag - Reddit, accessed on July 22, 2025, https://www.reddit.com/r/Rag/comments/1g5xpl6/which_model_should_i_use_to_embedding_150200/
32. Using Natural Language Inference to Improve Persona Extraction from Dialogue in a New Domain - arXiv, accessed on July 22, 2025, https://arxiv.org/html/2401.06742v1
33. Extracting Medical Information From Clinical Text With NLP - Analytics Vidhya, accessed on July 22, 2025, https://www.analyticsvidhya.com/blog/2023/02/extracting-medical-information-from-clinical-text-with-nlp/
34. Semantic Search: A Step-by-Step Guide with Python code | by Purnima M | Medium, accessed on July 22, 2025, https://medium.com/@purnima.msb/diy-semantic-search-a-step-by-step-guide-37e0b6df2a1f
35. A Comparative Study of PDF Parsing Tools Across Diverse Document Categories - arXiv, accessed on July 22, 2025, https://arxiv.org/html/2410.09871v1
36. Extract Text From a Multi-Column Document Using PyMuPDF in Python - Artifex Software, accessed on July 22, 2025,

https://artifex.com/blog/extract-text-from-a-multi-column-document-using-pymupdf-inpython